CSP dynamiques pour la génération de tests de systèmes réactifs

Christophe Junke, Benjamin Blanc

CSP dynamiques pour la génération de tests de systèmes réactifs

Christophe Junke, Benjamin Blanc

CEA LIST, Laboratoire de Sûreté des Logiciels
Boîte 65, F-91191 Gif-sur-Yvette France
prenom.nom@cea.fr

Résumé

GATeL est un outil de génération de tests dédié à la validation et la vérification de programmes réactifs synchrones. À partir d'une spécification décrivant un système réactif, de propriétés de son environnement ainsi que d'un objectif de test décrits en `Lustre`, les entrées successives du système menant à la réalisation de l'objectif de test sont générées. GATeL repose sur une interprétation en programmation logique par contraintes de `Lustre`, à l'aide de contraintes booléennes, arithmétiques et temporelles. Nous présentons dans cet article la propagation dynamique de contraintes avec stratégie en arrière de GATeL, qui construit de manière paresseuse les éléments du passé nécessaires à la réalisation d'un objectif de test.

1 Introduction

Le test logiciel [12] consiste à exécuter un logiciel dans le but de trouver ses défauts. C'est une activité dont l'automatisation pose plusieurs difficultés : sélectionner des cas de tests, trouver les entrées correspondantes et déterminer le succès ou l'échec à l'issue de l'exécution du test (oracle). Plusieurs outils proposent une approche automatique à base de programmation par contraintes pour les deux derniers points (INKA [9], BZ-TT [4], PathCrawler [16],Osmose [2]). À partir d'un problème de génération composé d'un programme et d'un objectif de test, ces outils produisent un problème de satisfaction de contraintes (CSP[1]) dans lequel les variables représentent les entrées, les domaines identifient les types de données, et les contraintes expriment les opérateurs du langage traité. Généralement, les langages de programmation permettent d'écrire des systèmes qui se traduisent par des CSP n'appartenant pas à une classe bien identifiée de problèmes de résolution. De plus, les domaines représentant les types numériques sont très grands et ne permettent pas de réutiliser les techniques mises au point pour des domaines finis. Les outils doivent alors combiner des solveurs spécifiques, et ce à un niveau suffisamment haut pour permettre le maximum de déductions entre les contraintes [10]. C'est pourquoi les équipes citées ont développé soit un solveur complet, soit une couche permettant de contrôler l'appel à des solveurs spécifiques.

GATeL. Nous présentons dans cet article l'interprétation en CSP du problème de génération de tests utilisée dans GATeL [11], un outil de validation et de vérification pour les systèmes réactifs. Un *système réactif* est un programme qui permet d'observer et de contrôler un environnement extérieur, physique ou logiciel, en interagissant avec lui de manière continue. L'approche flots de données synchrones [3] est un paradigme reconnu pour la modélisation de systèmes réactifs critiques : les langages synchrones disposent d'une sémantique formelle dans laquelle les calculs à chaque cycle sont déterministes en fonction des entrées successives du système. Cette approche facilite entre autres le développement de compilateurs certifiés, ce qui permet aux industriels de ces domaines, qui sont soumis à des normes strictes de validation et de vérification, de simplifier leurs processus de vérification. Nous nous intéressons au langage `Lustre` [5] ainsi qu'à son équivalent industriel, `Scade`[2]. Il existe un compilateur certifié (DO-178B) pour `Scade`, mais son utilisation n'est pas suffisante pour supprimer les

[1] Constraint Satisfaction Problem

[2] `http://www.esterel-technologies.com`

phases de tests car les normes imposent le test exhaustif des comportements décrits par la spécification (test fonctionnel). Pour chacun de ces comportement, GATeL peut générer des séquences d'entrées amenant le système de l'état initial jusqu'à la situation voulue.

Approche incrémentale. Pour les langages usuels (C, Java, B, etc.), une difficulté dans la génération des entrées est due à la présence d'itérateurs. Bien que ce type d'opérateur n'existe pas en `Lustre`, le langage possède implicitement une structure de contrôle itérative. En effet, il manipule des flots de données synchrones, qui peuvent être représentés à tout instant par des *séquences finies* de valeurs issues d'un état initial. Chaque flot est défini de manière mutuellement récursive par rapport à ses valeurs passées, et à celles d'autres flots. La résolution de contraintes correspond alors à une exploration synchronisée des définitions récursives de chaque flot apparaissant dans le problème.

GATeL propose une interprétation en Programmation Logique par Contraintes (PLC) de ce problème de génération de tests, ainsi qu'un solveur spécifique traitant les expressions de façon paresseuse, selon une stratégie de génération *en arrière*.À partir d'un CSP initial correspondant à l'interprétation de l'objectif de test, GATeL explore les passés possibles aboutissant à cet objectif. Une approche visant à construire des CSP statiques pour chaque longueur de séquence serait notoirement inefficace, dans la mesure où l'on se retrouverait dans un schéma *generate-and-test*. C'est pourquoi, afin de limiter la taille des séquences produites, l'exploration de nouveaux cycles est faite incrémentalement. Elle exploite les expressions temporelles relatives au contrôle du système sous test, qui contraignent les chemins d'exécution et le nombre de cycles nécessaires à l'élaboration d'un cas de test : la création de nouveaux cycles dans le passé ne se fait que lorsque le CSP courant l'impose. Cette exploration dynamique du passé est généralisée à celle des expressions booléennes, dont dépend aussi le contrôle, en les interprétant de manière paresseuse. Les expressions numériques sont quant à elles déléguées de manière classique à un solveur sur les domaines finis ou continus.

Plan. Nous présentons dans la section 2 les constructions du langage `Lustre`. La section 3 rappelle des définitions concernant les CSP dynamiques et donne les bases de notre interprétation. Nous montrons ensuite comment les contraintes de différentes natures sont construites efficacement (sec. 4). Des heuristiques de résolution appropriées, que nous abordons dans la partie 6, permettent de traiter des problèmes de génération de tests nécessitant des séquences très longues (en milliers de cycles). Nous présentons enfin dans la section 7 des extraits d'études de cas pour lesquels les performances obtenues sont mises en évidence.

2 Lustre

`Lustre` est un langage de modélisation utilisé principalement pour décrire des programmes réactifs de contrôle/commande. À chaque itération de la boucle réactive , les capteurs fournissent au système des valeurs d'entrées, qui servent à calculer les sorties destinées aux actionneurs. Les valeurs successives des entrées et sorties au cours du temps constituent des flots de données. Ainsi, toutes les expressions du langage dénotent une suite infinie de valeurs dans le temps : la constante `3` désigne le flot $(3, 3, \dots)$, l'expression `A+B` le flot $(A_0 + B_0, A_1 + B_1, \dots)$.

Approche synchrone. `Lustre` est un modèle de flots de données *synchrone* car l'ensemble de ces flots doivent avoir la même longueur à tout instant. Pour garantir cette propriété du modèle dans une application réelle, il est nécessaire de pouvoir borner le temps de calcul d'une itération de boucle pour l'ensemble des sorties du programme, afin d'obtenir le temps de cycle minimum du contrôleur. Les constructions présentes dans le langage assurent que cette borne existe pour chaque modèle `Lustre`.

Pour décrire des réactions temporelles complexes, `Lustre` permet de faire référence à des valeurs passées des flots de données, à travers l'opérateur de retard unitaire « `pre` ». C'est un opérateur unaire qui s'applique à un tout flot `X` de la manière suivante : `pre(X)` = $(\text{NIL}, X_0, X_1, X_2, \dots)$. L'opérateur `pre` réalise un décalage temporel d'un cycle. Puisqu'il n'existe pas de cycle antérieur au cycle initial, cet opérateur n'est pas défini au cycle 0, ce qui est figuré par le terme NIL dans l'exemple. Par extension, toute opération qui serait réalisée au cycle initial sur le flot `pre(X)` aurait un résultat indéterminé. Ce problème est levé par la deuxième primitive temporelle de `Lustre`, notée « `->` ». Cet opérateur binaire permet de définir la valeur initiale d'un flot comme suit : soient `A` et `B` deux flots `Lustre` de même type, alors `A -> B` = $(A_0, B_1, B_2, B_3, \dots)$.

Syntaxe du noyau `Lustre`. Un modèle `Lustre` est organisé en un ou plusieurs nœuds, définissant chacun un système d'équations où les flots de sortie sont exprimés en fonction de flots d'entrée par l'intermédiaire d'expressions arithmétiques, logiques ou temporelles. La figure 1 décrit la syntaxe abstraite d'un noyau de `Lustre`. Bien que celles-ci soient prises en

$$
\begin{array}{rcl}
\textit{expr} & ::= & \textit{bool} \mid \textit{integer} \mid \textsc{id} \\
& \mid & \texttt{pre}\ \textit{expr} \\
& \mid & \textit{expr}\ \texttt{->}\ \textit{expr} \\
& \mid & \textit{expr}\ \textsc{cmp}\ \textit{expr} \\
\textit{bool} & ::= & \textit{bool}\ \texttt{and}\ \textit{bool} \\
& \mid & \textit{bool}\ \texttt{or}\ \textit{bool} \\
& \mid & \texttt{not}\ \textit{bool} \\
& \mid & \texttt{true} \mid \texttt{false} \\
\textit{integer} & ::= & \textit{integer}\ \textsc{op}\ \textit{integer} \\
& \mid & \texttt{-}\ \textit{integer} \\
& \mid & \textsc{int} \\
\textsc{cmp} & ::= & = \mid <> \mid \leq \mid \geq \mid < \mid > \\
\textsc{op} & ::= & + \mid - \mid * \mid \textit{div} \mid \textit{mod}
\end{array}
$$

FIG. 1 – Syntaxe abstraite.

compte dans GATeL, nous ne nous intéressons pas ici aux constructions utilisant la notion d'horloge synchrone. De plus, nous ne traitons pas le cas des données de type réel dans cet article.

Pour permettre la déclaration d'objectifs de test, GATeL étend le langage par une directive spécifique, notée `reach`. Cette directive prend en argument n'importe quelle expression booléenne `Lustre` : l'expression `reach(Exp)` signifie que l'on souhaite construire des séquences pour lesquelles la condition `Exp` est vérifiée au dernier cycle de la séquence. Cette expression ayant la même expressivité que le langage, elle peut correspondre à la détection d'une alarme, ou encore décrire un scénario de test faisant transiter le système par des états particuliers au cours du temps.

```
node TEMPO(start,abort:bool;tempo:int(*!0..50!*))
returns (end_tempo: bool);
var counter, sel_tempo: int ;
    set: bool;
let
 set = start and not(abort) ->
       ( if   pre(set)
         then not(abort)
         else start and not(abort) );

 sel_tempo = if   start and not(abort)
             then tempo
             else 0 -> pre(sel_tempo);

 counter = if   set and not(start)
           then 1 -> pre(counter) + 1
           else 0;

 end_tempo = (counter > sel_tempo);
tel;
```

FIG. 2 – Le nœud `TEMPO`.

Exemple. La figure 2 montre la spécification d'une temporisation, représenté par le nœud `TEMPO`. Le système dispose de trois entrées et d'une sortie : lorsque la temporisation est démarée à l'aide de `start`, un compteur interne, `counter`, est incrémenté à chaque cycle d'une unité, jusqu'à ce qu'il atteigne ou dépasse la durée mémorisée au lancement du décompte, `sel_tempo`. La variable booléenne `end_tempo` prend alors la valeur `true`. À tout moment, la variable `abort` peut réinitialiser le compteur et arrêter le décompte. La figure 3 montre une trace d'exécution du nœud `TEMPO` illustrant le fonctionnement de la variable `set`. Celle-ci représente l'état de la temporisation, qui peut être active ou inactive. Elle vaut *true* à partir du moment où `start` passe à *true*, et reste dans cette état jusqu'à ce que `abort` devienne vrai.

Cycles	0	1	2	3	...
start	*false*	*true*	*false*	*false*	
abort	*false*	*false*	*false*	*true*	
E1	*true*	*true*	*true*	*false*	
E2	*false*	*true*	*false*	*false*	
E3		*false*	*true*	*true*	
E4		*true*	*true*	*false*	
set	*false*	*true*	*true*	*false*	

```
E1  = not(abort)
E2  = start and E1
E3  = pre(set)
E4  = if E3 then E1 else E2
set = E2 -> E4
```

FIG. 3 – Chronogramme d'exécution de `set`.

3 Construction dynamique du CSP

La génération de séquences de tests est un problème pour lequel le nombre d'itérations de la boucle réactive principale, nécessaire à la réalisation d'un objectif de test, n'est pas connu à l'avance. De plus, et bien qu'une analyse statique du programme sous test soit effectuée avant la génération de tests, il existe potentiellement des chemins de calculs très longs n'intervenant pas dans la réalisation d'un objectif de test donné. Afin de ne travailler que sur des contraintes liées au problème de génération, nous choisissons d'introduire ces contraintes de manière paresseuse, ce qui passe par la manipulation d'un CSP dynamique. Nous décrivons ici quelle interprétation de la notion de CSP dynamique nous partageons parmi celles qui ont été proposées dans la littérature. Tout d'abord, nous rappelons celle d'un CSP statique.

Définition 3.a. *CSP statique*

Un CSP statique P est donné par un triplet (V, D, C) représentant des ensembles de variables, de domaines et de contraintes.
À chaque variable v_i de l'ensemble $V = \{v_1, \ldots, v_n\}$, est associé un domaine D_i par une relation $dom(v_i) = D_i$.
Une contrainte de C est un couple $\langle R, \Sigma \rangle$ où R est une relation restreignant les domaines de valeurs des variables de l'ensemble $\Sigma \subseteq V$.
On appelle contrainte *atomique* $C = \langle R, \Sigma \rangle$ une contrainte pour laquelle l'ensemble des variables Σ est de cardinalité fixe. Par opposition, une contrainte est dite *globale* si la cardinalité de Σ n'est pas connue.

Il y a deux manières de considérer les CSP dynamiques, selon que les modifications sont faites depuis l'intérieur ou l'exterieur de la procédure de recherche de solutions [15]. L'évolution est externe lorsque le caractère dynamique reflète des modifications d'un problème selon son environnement ou une interaction particulière. Par exemple, les problèmes de gestion de ressources sont dynamiques car les données du problèmes peuvent changer au cours du temps (changement d'une activité dans un planning, retrait ou ajout d'une ressource, ...). Dans ce cas, un problème intéressant lié à ces problèmes dynamiques consiste à garantir la stabilité des solutions tout en étant efficace, et cela passe par la réutilisation de solutions d'une modification à une autre du problème [14]. Il existe ensuite deux types d'évolution interne de CSP dynamiques. Un premier type consiste à ne considérer qu'un certain nombre de variables du CSP à un instant donné en fonction de l'instanciation partielle [13]. Le second type d'évolution interne apparait lorsque le CSP manipule des contraintes globales pouvant elles-mêmes introduire d'autres contraintes (atomiques ou globales). Le moteur d'INKA traduit par exemple les boucles *while* par une telle contrainte, car la valeur de vérité de la boucle est une variable du CSP, et son instanciation peut introduire de nouvelles variables et contraintes correspondant au corps de sa boucle. C'est ce type d'évolution que nous mettons en oeuvre dans GATeL, pour gérer l'aspect temporel et logique du flot de contrôle.

Définition 3.b. *CSP dynamique*

Si $P = (V, D, C)$ est un CSP, alors $P' = (V', D', C')$ tel que $V \subseteq V'$, $D'_v \subseteq D_v$ $(\forall v \in V)$, et $C \subseteq C'$ est une *restriction* de P.
Un CSP dynamique est une succession de problèmes P_i telle que chaque P_i soit une restriction de P_{i-1}.

Ainsi, un problème de satisfaction de contraintes est dynamique lorsqu'il existe une suite de transitions entre des CSP statiques [7]. Nous nous intéressons ici uniquement aux transitions provoquant des restrictions successives. Dans la suite de cette section, nous décrivons ce que représentent les ensembles (V, D, C) dans le cadre de notre problème de génération de tests.

Variables. Nous introduisons une matrice $\mathcal{M}$ de variables logiques, où chaque ligne représente un flot nommé du modèle `Lustre` (entrée ou sortie), et le nombre de colonnes est variable et correspond au nombre de cycles nécessaires pour atteindre l'objectif. Le point essentiel de notre interprétation est d'introduire les variables logiques dynamiquement, en fonction des besoins. Afin de permettre une résolution du problème de contraintes de façon efficace, toutes les variables d'une colonne de la matrice ne sont pas forcément présentes à un instant donné. Une case de $\mathcal{M}$ ne contient une variable logique que si la variable de flot correspondante a été introduite dans le CSP au cycle considéré ; elle est vide sinon.

Pour coder l'aspect temporel de `Lustre`, deux variables logiques supplémentaires sont introduites. La première, notée C_{max}, exprime le numéro du plus grand numéro de cycle connu dans le CSP courant. En effet, l'opérateur `pre` introduisant une récursion naturelle pour la plupart des définitions de flot, le nombre de cycles nécessaire pour atteindre un objectif est lui-même dépendant du problème. GATeL créant les séquences en arrière, depuis le cycle final vers le cycle initial, C_{max} est initialisé à 0 et incrémenté à chaque fois qu'un nouveau cycle est nécessaire à la résolution. Ainsi les cycles sont numérotés dans l'ordre croissant depuis le cycle final vers le cycle initial. À cause de la progression en arrière, on ne sait pas à l'avance, lorsqu'un cycle est créé, si celui-ci est le cycle initial de la séquence. La seconde variable, notée *Statut*, symbolise cette information pour le plus grand numéro de cycle connu (les autres cycles étant forcément non initiaux). Ce statut intervient dans le traitement de l'opérateur « -> » : cet opérateur est équivalent à son membre gauche quand le statut est initial, et à son membre droit sinon.

L'ensemble V des variables présentes à un instant donné dans le CSP est l'union des variables logiques de $\mathcal{M}$, des deux variables temporelles, et potentiellement de variables supplémentaires permettant de décomposer les expressions complexes vers des contraintes atomiques. Les variables logiques sont introduites à l'aide de la primitive $fresh(D)$, qui retourne une variable libre V telle que $dom(V) = D$.

Domaines. Chaque variable logique de V est attribuée à un domaine $dom(V)$ correspondant à sa déclaration de type. Les variables booléennes sont attribuées dans un intervalle de valeurs symboliques $[true, false]$. Le domaine d'une variable entière est une union d'intervalles croissants $[min_1..max_1] \cup ... \cup [min_n..max_n]$, dont les bornes inférieures et supérieures peuvent être

aussi grandes que nécessaire (en particulier pour coder des entiers sur 32, 64 bits ou autre). Ce domaine permet de capturer les inconsistances proche de zéro dans le cas de multiplications. Si $A = 3 * B$ et $B \neq 0$, alors le domaine de A ne peut pas contenir : -2, -1, 1, 2. Ces trous dans les intervalles peuvent être ensuite décalés en fonction des opérations effectuées. La variable logique *Statut* prend des valeurs dans le domaine symbolique $\{initial, non\_initial\}$, et la variable C_{max} est bornée par un paramètre global pour stopper le déroulement du passé dans le cas de récursions non bornées.

Contraintes. Chaque classe de contraintes est généralement associé à un algorithme de filtrage, ou propagateur, qui se charge de retirer les valeurs des domaines des variables contraintes, tout en garantissant qu'aucune solution n'est écartée du CSP. Nous distinguons deux types de contraintes dans GATeL.

Tout d'abord, les contraintes atomiques, non spécifiques à GATeL, qui décrivent des relations arithmétiques ($+, *, -, \leq$, etc.). Pour chaque contraite numérique $\langle R, \Sigma \rangle$, l'algorithme de filtrage définissant la relation R est une consistance de bornes sur les unions d'intervalles des variables de Σ. Ces contraintes portent uniquement sur une variable résultat et deux variables arguments.

Par ailleurs, nous considérons une contrainte propre à notre méthode de génération, qui met en œuvre l'aspect paresseux de l'approche. Cette contrainte est de la forme $propage(Res, Exp, C)$ où *Res* représente la valuation de l'expression `Lustre` *Exp* au cycle C. L'algorithme de filtrage définissant la relation R de cette contrainte est défini inductivement selon l'opérateur de tête du terme *Exp*. Lorsque ce terme est donné par un opérateur arithmétique, la contrainte délègue sa relation à une contrainte atomique. Lorsqu'il s'agit d'un opérateur booléen ou temporel, si *propage* a suffisemment d'information sur l'expression *Exp*, elle peut introduire de nouvelles contraintes correspondant aux sous-termes de *Exp*. Sinon, la contrainte est retardée jusqu'à ce que le domaine d'une des variables de son ensemble Σ soit réduit. La contrainte *propage* est globale car l'ensemble Σ de variables sur lequel travaille l'algorithme de filtrage n'est pas complètement défini tant que toutes les contraintes atomiques correspondant au terme *Exp* n'ont pas été introduites. Cet ensemble Σ est calculé par une fonction auxiliaire *eval* qui analyse les termes pour retrouver les variables de $\mathcal{M}$, sous la portée de *Exp*, qui sont soit déjà propagées soit des entrées, ou bien la variable *Statut* dans certains cas. La fonction *eval* est décrite dans la section 5.

Considérons la propagation de l'expression « `(E > 0) and S` », où `E` est une entrée et `S` est une sortie, dont le résultat vaut `true`, au cycle 0. Le résultat de l'expression étant vrai, l'opérateur `and` peut introduire deux contraintes correspondant à ses sous-termes :

$$\left\{ \begin{array}{l} \langle propage(true, \texttt{(E > 0)}, 0), \emptyset \rangle, \\ \langle propage(true, \texttt{S}, 0), \emptyset \rangle \end{array} \right\}$$

La propagation de la première contrainte va être déléguée vers la contrainte arithmétique « > » avec la variable E_0 introduite dans $\mathcal{M}$, correspondant à l'entrée `E` au cycle 0. Si de plus `S1` est définie par l'expression « `E' or S'` », le point fixe par propagation de contraintes du CSP est :

$$\left\{ \begin{array}{l} \langle E_0 > 0, \{E_0\} \rangle \\ \langle propage(true, \texttt{(E' or S')}, 0), \{E'_0\} \rangle \end{array} \right\}$$

avec $\mathcal{M}(S, 0) = true$.

Notons qu'aucune branche de la disjonction n'est ajoutée sous forme de contrainte. Supposons que lors de la recherche de solutions, la variable E'_0 soit instanciée à *false*. La contrainte de disjonction est de nouveau filtrée, ce qui raffine le système de contraintes en introduisant la définition de `S'` au cycle 0, puisqu'elle est nécessaire. Ce mécanisme consistant à ne pas introduire systématiquement toutes les contraintes à partir des sous-termes est à la base de l'aspect dynamique de GATeL.

4 Définition inductive

GATeL est implémenté en PLC dans l'environnement `ECLiPSe` [1]. Les règles associées à la sémantique opérationnelle de la contrainte globale *propage* sont présentées dans cette section, sur quelques cas représentatifs. L'opérateur « := » représente l'affectation, par opposition aux tests d'égalité et d'inégalité structurels ($=, \neq$) et à l'unification ($\equiv$). L'opérateur *suspend* permet de placer une contrainte en attente de la réduction du domaine d'une de ses variables.

Variable de flot. Lorsque la valeur d'une variable de flot *id* doit être connue à un cycle donnée, deux cas se présentent (*cf.* règle ci-après). Soit la variable a déjà été propagée, auquel cas sa valeur dans $\mathcal{M}$ est unifiée avec la variable *Res* (ligne 4), soit il faut la propager, et une nouvelle variable logique est créée dans la matrice avec le domaine adéquat, noté Dom_{id} (ligne 5). S'il s'agit d'une sortie, l'expression de définition Exp_{id} de cette variable est propagée avec la variable de résultat *Res* (ligne 7).

```
1  propagate(Res, id, cycle)
2  ─────────────────────────────────────────
3  si  M(id, cycle) ≠ ∅
4  alors  Res ≡ M(id, cycle)
5  sinon  Res := fresh(Dom_id)
6         M(id, cycle) := Res
7         propage(Res, Exp_id, cycle)
```

Opérateur and. Dans le cas où le résultat est *true*, deux contraintes sont introduites, une pour chacun de ses arguments. Sinon, une évaluation partielle du premier argument est effectuée à l'aide de l'évaluateur d'expressions *eval* (ligne 7). Cette évaluation calcule un éventuel résultat instancié pour cet argument et remonte un ensemble de variables rencontrées au cours de cette évaluation (voir section 5). La principale différence entre *eval* et *propage* est que les autres contraintes du CSP ne peuvent pas se réveiller pendant le calcul du premier. Il s'agit donc d'une évaluation locale. Si le résultat de l'évaluation est *true*, alors deux contraintes sont introduites (lignes 9 et 10). Si l'évaluation est *false*, le résultat de la contrainte de départ est unifié à *false* (ligne 14), provoquant le réveil des contraintes en attente sur une évolution de cette variable. On effectue le cas échéant la même opération pour le second argument. Si aucune réduction n'a pu être effectuée, la contrainte se suspend en attente de l'évolution d'une des variables remontées par l'évaluation partielle, ou du résultat (ligne 29).

```
1  propage(Res, ExpA and ExpB, cycle)
2  ─────────────────────────────────────────
3  si  Res = true
4  alors  propage(true, ExpA, cycle)
5         propage(true, ExpB, cycle)
6  sinon
7    (R_A, V_A) := eval(ExpA, cycle)
8    si  R_A = true
9    alors  propage(true, ExpA, cycle)
10          propage(Res, ExpB, cycle)
11   sinon
12     si  R_A = false
13     alors  propage(false, ExpA, cycle)
14            Res ≡ false
15     sinon
16       (R_B, V_B) := eval(ExpB, cycle)
17       si  R_B = true
18       alors  propage(Res, ExpA, cycle)
19              propage(true, ExpB, cycle)
20       sinon
21         si  R_B = false
22         alors  Res ≡ false
23                propage(false, ExpB, cycle)
24         sinon
25           si  Res = false
26           alors  Σ := V_A ∪ V_B
27           sinon
28             Σ := {Res} ∪ V_A ∪ V_B
29             suspend(propage(Res, ExpA and ExpB, cycle), Σ)
```

De la même manière, la contrainte *propage* d'un opérateur conditionnel ne propage pas systématiquement ses arguments. Elle attend de connaitre la valeur de vérité de sa condition auparavant. Néanmoins, l'expression correspondant à la condition est toujours propagée.

Opérateur `pre`. Par hypothèse d'induction, et de par la bonne initialisation des programmes `Lustre` sous test [6], nous supposons que lorsque la contrainte *propage* arrive sur un opérateur `pre` à un cycle c, la propriété $C_{max} > c$ est toujours vérifiée. La contrainte traverse cet opérateur en propageant l'expression retardée au cycle précédent :

```
1  propage(Res, pre(Exp), cycle)
2  ─────────────────────────────────────────
3    propage(Res, Exp, cycle + 1)
```

Opérateur d'initialisation. Cet opérateur transcrit l'opérateur « -> » de `Lustre`. Si le plus grand numéro de cycle connu est supérieur au cycle auquel on veut propager cet opérateur, le membre droit est directement propagé (ligne 4) car le cycle courant n'est pas le cycle initial. Sinon, on fait intervenir le statut d'initialité : s'il est connu, alors le sous-terme correspondant est propagé (lignes 7 et 10) ; s'il est variable, une évaluation partielle des sous-termes est effectuée : pour le membre gauche (ligne 12), si le résultat évalué R_I est incompatible avec le résultat attendu *Res*, alors le statut est unifié à *non_initial*, ce qui provoque des réveils pour toutes les contraintes potentiellement en attente sur ce statut ; le numéro du plus grand cycle connu est incrémenté et une nouvelle variable de statut est créée. Enfin, le sous-terme de droite est propagé. Pour le membre droit (ligne 20), si le résultat évalué R_P est incompatible avec *Res*, le statut actuel est unifié à *initial*, provoquant aussi d'éventuels réveils. Le membre gauche est ensuite propagé. Si aucune des évaluations n'est incompatible, alors la contrainte se suspend sur les variables remontées lors de l'évaluation, le résultat et le statut du cycle courant (ligne 28).

```
1  propage(Res, ExpI -> ExpP, cycle)
2  ─────────────────────────────────────────
3  si  C_max > cycle
4  alors  propage(Res, ExpP, cycle)
5  sinon
6    si  Statut = initial
7    alors  propage(Res, ExpI, cycle)
8    sinon
9      si  Statut = non_initial
10     alors  propage(Res, ExpP, cycle)
11     sinon
12       (R_I, Σ_I) := eval(ExpI, cycle)
13       si  dom(R_I) ∩ dom(Res) = ∅
14       alors
15         Statut ≡ non_initial
16         Statut := fresh({initial, non_initial})
17         C_max := C_max + 1
18         propage(Res, ExpP, cycle)
19       sinon
20         (R_P, Σ_P) := eval(ExpP, cycle)
```

```
21          si  dom(R_P) ∩ dom(Res) = ∅
22          alors
23            Statut ≡ initial
24            propage(Res, ExpI, cycle)
25          sinon
26            Σ := Σ_I ∪ Σ_P ∪ {Res, Statut}
27            Cstr := propage(Res, ExpI-> ExpP, cycle)
28            suspend(Cstr, Σ)
```

Opérateurs arithmétiques. Nous prenons exemple ici de l'opérateur « + » des entiers. Lorsque la contrainte *propage* rencontre un opérateur arithmétique, elle délègue sa gestion à des contraintes spécifiques capables de réduire les domaines de ses variables selon une consistance de bornes appliquée à l'union des intervalles (ici *plus*, ligne 7). Une gestion paresseuse de ces contraintes ne serait pas efficace car la contrainte *propage* n'aurait que peu d'occasions de réduire des sous-termes (uniquement sur les éléments neutres ou absorbants). De plus, ces contraintes arithmétiques doivent pouvoir travailler au plus vite pour influencer les décisions booléennes ou temporelles. Les contraintes arithmétiques ne travaillant que sur des variables attribuées, les arguments sont systématiquement propagés (lignes 5 et 6). Des variables intermédiaires sont ainsi créées avec le type correspondant au résultat (ici des unions d'intervalles, dont le domaine initial est noté $\mathcal{I}$).

```
1  propage(Res, ExpA + ExpB, cycle)
2  ──────────────────────────────────
3  R_A := fresh(I)
4  R_B := fresh(I)
5  propage(R_A, ExpA, cycle)
6  propage(R_B, ExpB, cycle)
7  plus(Res, R_A, R_B)
```

5 Évaluation partielle

L'évaluation partielle des expressions de flots `Lustre` est une fonction dépendant de la matrice $\mathcal{M}$ et du cycle courant. Elle s'applique aux termes des expressions de flots où les variables `Lustre` ne sont pas forcément connues au moment de l'évaluation. Si l'expression de flot peut-être évaluée à un cycle, elle retourne la valeur du flot selon la sémantique `Lustre`. Lorsque des sous-expressions ne peuvent pas encore être évaluées, elle retourne l'ensemble des variables logiques l'évaluation n'a pas été possible (déjà propagées, ou bien de nouvelles variables associées à des entrées on encore propagées). Cela permet au propagateur d'attendre l'instanciation d'au moins une de ces variables avant de tenter à nouveau l'évaluation.

La règle suivante montre l'évaluation de l'opérateur and. On retrouve la sémantique paresseuse de cet opérateur dans la définition. Si aucune instanciation n'a pu être effectuée, la fonction retourne l'ensemble des variables remontées par l'évaluation des sous-termes.

```
1  eval(Exp_1 and Exp_2, cycle)
2  ──────────────────────────────────
3  (R_1, Σ_1) := eval(Exp_1, cycle)
4  si R_1 = true
5  alors retourner eval(Exp_2, cycle)
6  sinon si R_1 = false
7         alors retourner (false, ∅)
8         sinon (R_2, Σ_2) := eval(Exp_2, cycle)
9               si R_2 = true
10              alors retourner (R_1, Σ_1)
11              sinon si R_2 = false
12                    alors retourner (false, ∅)
13                    sinon Val := fresh({false, true})
14                          retourner (Val, Σ_1 ∪ Σ_2)
```

Les variables logiques issues de l'évaluation de $Exp1$ et de $Exp2$ correspondent à des variables de flots dont la définition a déjà été propagée. Les autres opérateurs sont construits de la même manière. La règle d'évaluation d'une variable de flot *id* à un cycle est définie ainsi :

```
1  eval(id, cycle)
2  ──────────────────────────────────
3  si M(id, cycle) = ∅
4  alors R := fresh(Dom_id)
5         retourner (R, ∅)
6  sinon retourner (M(id, cycle), M(id, cycle))
```

6 Génération de séquences de tests

La génération de séquences de tests consiste à résoudre un CSP initial défini par l'unique contrainte *propage* représentant l'objectif de test. Son résultat vaut *true*, le terme est l'expression booléenne spécifiée par la directive `reach`, et le cycle vaut 0.

Exemple. Reprenons l'exemple de la figure 2 avec l'objectif « `reach end_tempo` ». La seule contrainte du CSP initial est donc propage(true, `end_tempo`, 0). La définition du flot `end_tempo` est « `counter > sel_tempo` ». L'algorithme de filtrage introduit les définitions de `counter` et de `sel_tempo` au cycle 0, en les associant à des variables logiques que l'on note respectivement C_0 et Sel_0. L'inégalité est déléguée à la contrainte $C_0 > Sel_0$.

La variable `counter` est définie par une expression conditionnelle. Ici, la propagation de cette expression ne peut pas savoir quelle branche correspond au résultat. Dans un premier temps, seule la condition est propagée : la variable If_c représente le résultat de la condition « `set and not(start)` ». La propagation de l'opérateur s'arrête ici, car il n'y a pas assez d'information au niveau du terme « and ». Une évaluation partielle de ses sous-termes est faite pour savoir quelles sont les variables logiques déjà propagées dont dépend l'expression. Dans notre cas, seule la variable d'entrée $Start_0$ est remontée, la variable `set` n'étant pas encore propagée au cycle 0.

L'expression and dépend de l'ensemble de variables $\Sigma_0 = \{If_c, Start_0\}$. On note P_0 la contrainte suivante :

$$P_0 = \langle propage(If_c, \texttt{ set and not(start) }, 0), \Sigma_0 \rangle$$

Par ailleurs, l'expression conditionnelle est elle aussi traduite en une contrainte. Pour cela, *eval* est appliquée à chaque branche du `if` pour remonter un ensemble de variables logiques : les ensembles $V_{then} = \{Statut\}$ et $V_{else} = \emptyset$. La contrainte P_1 suivante est suspendue en attente d'une réduction de domaines sur les variables de $\Sigma_1 = \{C_0, If_c, Statut\}$:

$$\begin{aligned} & P_1 = \langle propage(C_0, Exp_1, 0), \Sigma_1 \rangle \\ avec \quad & Exp_1 = \texttt{if(}If_c\texttt{,1->pre(counter)+1,0)} \end{aligned}$$

La propagation de la définition de `sel_tempo` est similaire à celle de `counter`, et va provoquer la suspension des deux contraintes P_2 et P_3 suivantes :

$$\begin{aligned} & P_2 = \langle propage(If_s, Exp_2, 0), \Sigma_2 \rangle \\ & P_3 = \langle propage(Sel_0, Exp_3, 0), \Sigma_3 \rangle \\ avec \quad & Exp_2 = \texttt{start and not(abort)}, \\ & Exp_3 = \texttt{if(}If_s\texttt{,tempo,0->pre sel\_tempo)} \\ & \Sigma_2 = \{If_s, Start_0, Abort_0\}, \\ et \quad & \Sigma_3 = \{If_s, Sel_0, Tempo_0\}. \end{aligned}$$

De plus lors de l'évaluation de ses branches, la fonction *eval* a montré que le domaine de Sel_0 était compris entre 0 et 50, à partir de la restriction donnée à `tempo` dans le modèle. La contrainte d'inégalité représentant l'objectif impose la réduction de la borne inférieure du domaine de C_0 à 1, ce qui provoque le réveil de la contrainte P_1. Lors de la propagation, on constate par évaluation partielle que le domaine de l'expression correspondant à la branche *else* est incompatible avec celui du résultat. Cela impose que la condition If_c soit vraie, ce qui a plusieurs conséquences : Tout d'abord, puisque la condition est vraie, l'expression « `1 -> pre(counter) + 1` » est propagée et unifiée avec le résultat C_0. Une nouvelle contrainte P_4 est ajoutée. Ensuite, la variable If_c étant instanciée avec la valeur *true*, la contrainte P_0 est réveillée et les deux branches de la conjonction sont alors propagées. La première branche va imposer que Set_0 soit vraie puis suspendre sa définition en attente d'informations supplémentaires concernant entre autre la variable $Start_0$; la seconde branche impose justement que $Start_0$ soit faux. Cette dernière information est donc reprise par la contrainte de définition de Set_0. Puisque $Start_0$ est fausse, la fonction *eval* détermine que l'expression `start and not(abort)`, en membre gauche de l'opérateur d'initialisation porté par Set_O, doit être fausse. Comme Set_0 est vraie, il n'est pas possible que le statut du cycle 0 soit initial. La variable *Statut* est donc instanciée à *non_initial*, et C_{max} repoussé au cycle 1. Enfin, ce changement de statut a pour effet de réveiller P_4 et d'introduire la définition de `counter` au cycle 1.

La propagation continue jusqu'à atteindre un point fixe composé de 8 contraintes portant du cycle 0 au cycle 2. GATeL choisit alors une variable et effectue une étape de résolution afin de réveiller certaines contraintes.

Heuristiques de résolution. La résolution permet de faire avancer la génération de séquences de tests lorsqu'un point fixe est atteint, en instanciant une variable logique avec une valeur de son domaine. Le choix de la variable se fait en plusieurs étapes :

1. Récupérer les variables booléennes qui ont le plus de contraintes en attente, et parmi celles-ci, en choisir une au plus grand numéro de cycle.
2. Comparer ce nombre de contraintes avec le nombre de contraintes en attente sur la variable *Statut*.
3. Si ce deuxième nombre est plus grand, chercher à fermer le passé en instanciant le statut à *initial*. Si cette instanciation échoue, le statut est instancié à *non_initial* et la procédure recommencera après le point fixe de propagation qui suivra.
4. Sinon, instancier la variable booléenne de manière aléatoire dans son domaine.
5. Si aucune variable booléenne ni de statut n'est présente dans les contraintes, la procédure choisit une variable entière correspondant à une entrée et l'instancie de manière aléatoire dans son domaine.

La procédure cherche donc en priorité à faire avancer la partie paresseuse de la propagation avant de s'intéresser à la partie arithmétique. En pratique, les problèmes auxquels on se réfère sont très sous-contraints. Cette partie arithmétique de la procédure est assez rarement appelée.

7 Performances

Nous présentons dans cette section quelques résultats sur des études de cas mettant en évidence l'intérêt de la propagation paresseuse. Les comparaisons sont effectuées entre la version courante de GATeL implémentant les définitions précédentes, et une version modifiée dans laquelle les opérateurs booléens propagent systématiquement leurs arguments. Ces expériences ont été réalisées sur un processeur Intel P4 de 3.6GHz avec 1Go de RAM.

Stratégie	Paresseuse	Systématique
Temps d'execution	**2673**	**17941**
Propagation (ms)	2519	8691
Résolution (ms)	154	9250
Nb. de contraintes	**2343**	**2508**

FIG. 4 – Propagation initiale pour `mode_de_fonct`.

Mesures	Mémoire (Ko)	Temps (ms)
1	5443	169
2	5443	170
3	5443	139
4	5444	169
5	5443	160
6	5443	120
Moyenne	5443,17	154,5
Ecart relatif	0,01%	13,33%

FIG. 5 – Résolution de `mode_de_fonct`, cas paresseux.

Mesures	Mémoire (Ko)	Temps (ms)
1	4773	310
2	114356	18330
3	116270	18640
4	4752	300
5	9750	259
6	114780	18170
Moyenne	60826,17	9249,67
Ecart relatif	98,03%	106,12%

FIG. 6 – Résolution de `mode_de_fonct`, cas systématique.

`mode_de_fonct` Cet exemple est extrait d'un système de protection d'un réacteur nucléaire. Il détermine le mode de fonctionnement : normal, dégradé ou arrêt, selon l'observation d'un taux de comptage de neutrons. L'objectif de test est d'observer un passage en arrêt d'urgence. Pour se produire, celui-ci nécessite 1000 cycles de calcul en maintenant certaines entrées dans des intervalles de variations définis.

Le tableau 4 illustre le temps d'exécution de la propagation initiale de l'objectif de test ainsi que le nombre de contraintes résultant pour les deux stratégies. Les tableaux 5 et 6 donnent les temps de résolution nécessaire pour découvrir les 1000 cycles, ainsi que les écarts-types relatifs constatés. Les temps de calcul et la mémoire occupée sont en moyenne nettement meilleurs dans le cas paresseux, et très stables. Notons cependant que certaines exécutions donnent de meilleurs résultats mémoire dans le cas systématique, mais avec une variabilité plus grande. La grande variabilité est liée à l'imperfection des heuristiques en présence d'un grand nombre de contraintes.

`A33_34` Cet exemple est extrait d'un benchmark de systèmes de protection [8]. Il contient des voteurs sur des entrées répliquées respectant des lois de variations temporelles (contraintes de pente) en observant des dépassements de seuils à hystéresis (haut ou bas), pour positionner des conditions de contrôle. L'objectif de test est d'observer deux dépassements de seuils sur les seconds min de 2 groupes de 4 entrées répliquées. Compte-tenu des contraintes de pente, il nécessite la création d'au moins 80 cycles pour atteindre ces seuils.

Le tableau de la figure 7 montre les temps observés pour la propagation initiale de contraintes. Dans le cas de la stratégie systématique, 12443 contraintes ont été propagées en approximativement 40 secondes. La version paresseuse propage quant à elle 8176 contraintes en 4 secondes. Bien que la première dispose de plus de contraintes, aucune des tentatives de résolution n'a permis d'obtenir une solution en un temps raisonnable (abandons au bout de 15 minutes). Le tableau 8 montre les mesures de l'occupation mémoire et du temps d'execution pour l'étape de résolution avec propagations paresseuses. Sur 12 mesures, 2 n'ont pas abouti à un résultat en un temps raisonnable. La moyenne et l'écart-type relatif sont calculés sur les 10 essais réussis.

`CruiseStateMgt` Cet exemple est le codage en flot de données d'un automate de contrôle d'un système de régulation de vitesse. Un objectif de test intéressant est de montrer que certaines sorties de cet automate sont incompatibles. Dans ce cas, GATeL doit montrer qu'il n'existe aucune séquence aboutissant à un tel objectif. Pour cela, nous limitons le nombre de cycles d'exploration à 5. Les résultats présentés dans le tableau 9 donnent le temps nécessaire à l'exploration complète de l'espace de recherche sans obtenir de solution pour chaque stratégie. On observe que la version systématique est nettement plus rapide dans ce parcours. En effet, les flots booléens définissant cet automate sont très fortement corrélés entre eux (partage de sous-expressions communes), ce qui peut être exploité par la stratégie systématique pour réduire l'espace de recherche.

8 Conclusion

Nous avons présenté le mécanisme paresseux d'introduction de contraintes de GATeL. Ce mécanisme s'applique aux opérateurs temporels et booléens pour minimiser le nombre de contraintes et de variables du CSP à un instant donné. Le CSP évolue dynamiquement par restrictions successives dues à l'introduction de nouvelles contraintes au fur et à mesure de la génération de séquences. Nous obtenons des résultats

Mesures	Systématique	Paresseuse
1	40409	3740
2	40409	3710
3	44689	3750
4	40320	3700
5	40299	3700
Moyenne	41225,2	3720
Ecart relatif	4,70%	0,63%

FIG. 7 – Propagation initiale de `A33_34` (ms).

Mesures	Mémoire (Ko)	Temps (ms)
1	37348	108889
2	38782	87809
3	38298	96979
4	31963	40009
5	41016	117609
6	39979	75509
7	38009	85369
8	31309	32399
9	32079	56399
10	43675	103629
Moyenne	37245,8	80460
Ecart relatif	11,21%	36,18%

FIG. 8 – Résolution de `A33_34` dans le cas paresseux.

stables et efficaces pour les exemples industriels dont nous disposons.

Les évolutions récentes du langage Scade visent à introduire la notion d'automate de manière intégrée au paradigme flots de données synchrones. Les automates sont traduits en utilisant de façon intensive la notion d'horloge synchrone sur des types énumérés. Cette extension est en cours d'intégration dans GATeL.

Stratégie	Paresseuse	Systématique
1	333459	77760
2	314890	71400
3	325440	73830
4	309530	74190
Moyenne	320829,75	74295
Ecart relatif	0,03%	0,04%

FIG. 9 – Résolution pour `CruiseStateMgt` (ms).

Références

[1] K. Apt and M. Wallace. *Constraint Logic Programming using ECLiPSe*. Cambridge University Press, 2006.

[2] S. Bardin and P. Herrmann. Structural testing of executables. In *ICST*, pages 22–31. IEEE Computer Society, 2008.

[3] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1) :64–83, 2003.

[4] F. Bouquet, B. Legeard, and F. Peureux. CLPS-B : A constraint solver to animate a B specification. *International Journal on Software Tools for Technology Transfer, STTT*, 6(2) :143–157, August 2004.

[5] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John A. Plaice. LUSTRE : A declarative language for programming synchronous systems. pages 178–188. ACM Press New York, NY, USA, 1987.

[6] J-L. Colaço and M. Pouzet. Type-based Initialization Analysis of a Synchronous Data-flow Language. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(3) :245–255, August 2004.

[7] Rina Dechter and Avi Dechter. Belief maintenance in dynamic constraint networks. In *7th Conference On Artificial Intelligence*, 1988.

[8] Jean Gassino, Pascal Régnier, Bruno Marre, and Benjamin Blanc. Criteria and Associated Tool for Functional Test Coverage of Safety Critical Software. In *Proceedings of 4th ANS International Topical Meeting on Nuclear Plant Instrumentation, Controls and Human-Machine Interface Technologies (NPIC HMIT 2004)*.

[9] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. *ACM SIGSOFT Software Engineering Notes*, 23(2) :53–62, 1998.

[10] Arnaud Gotlieb, Bernard Botella, and Mathieu Watel. Inka : Ten years after the first ideas. In *19th International Conference on Software and Systems Engineering and their Applications (ICSSEA'06)*, 2006.

[11] Bruno Marre and Benjamin Blanc. Test Selection Strategies for Lustre Descriptions in GATeL. *Electronic Notes in Theoretical Computer Science*, 111 :93–111, 2005.

[12] Aditya P. Mathur. *Foundations of Software Testing*. Pearson Education, 2008.

[13] S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 1990.

[14] Gérard Verfaillie and Narendra Jussien. Constraint solving in uncertain and dynamic environments – a survey. *Constraints*, 10(3) :253–281, 2005.

[15] Gérard Verfaillie and Thomas Schiex. Solution reuse in dynamic constraint satisfaction problems. In *12th Conference On Artificial Intelligence*, 1994.

[16] Nicky Williams, Bruno Marre, and Patricia Mouy. On-the-fly generation of k-path tests for C functions. In *19th International Conference on Automated Software Engineering, 2004.*, pages 290–297, 2004.